A Formal Verification Methodology for IP-based Designs

Daniel Karlsson, Petru Eles, Zebo Peng
IDA, Linköpings universitet
{danka, petel, zebpe}@ida.liu.se

Abstract

This paper proposes a formal verification methodology which smoothly integrates with component-based system-level design, using a divide and conquer approach. The methodology assumes that the system consists of several reusable components, each of them already verified by their designers and which are considered correct under the assumption that the environment satisfies certain properties assumed by the component. What remains to be verified is the glue logic inserted between the components. Each such glue logic is verified one at a time using model checking techniques.

Experiments, performed on a real-life example (mobile telephone), demonstrating the efficiency and intuitivity of the methodology, are moreover thoroughly presented. Three different properties have been verified on one part of the system.

1. Introduction

It is a well-known fact that we increasingly often interact with electronic devices in our everyday lives. Such electronic devices are, for instance, cell phones, PDAs and portable music devices such as Mp3-players. Moreover, other, traditionally mechanical, devices are becoming more and more computerised. Examples of such devices are cars or washing machines. Several such devices are in addition highly safety critical, such as aeroplanes or medical equipment. In fact, in 1999, 99% of all microprocessors were used in the type of systems mentioned above (embedded systems). Only the remaining 1% was used in general purpose computers [1]. This situation indicates the big importance of embedded systems.

Obviously, it is both very error-prone and time-consuming to design such complex systems. At the same time there is a strong economical incentive to decrease the time-to-market.

In order to manage the design complexity and to decrease the development time, designers usually resort to reusing existing components (so called IP blocks) so that they do not have to develop certain functionality themselves from scratch. These components are either developed in-house by the same company or acquired from specialised IP vendors [2, 3].

Formal verification tools analyse the system model, captured in a particular design representation, to find out whether it satisfies certain properties. In this way, the verification tool can trap many design mistakes at early stages in the design.

Since the trend is that systems are built more and more with reusable components, it becomes increasingly important to develop verification methodologies which can effectively cope with this situation and take advantage of it.

There are several aspects which make this task difficult. One is the complexity of the systems, which makes simulation based techniques very time consuming. On the other hand, formal verification of such systems suffers from state explosion. However, it can often be assumed that the design of each individual component has been verified [4] and can be supposed to be correct. What remains to be verified is the interface logic and the interaction between components. Such an approach can handle both the complexity aspects (by a divide and conquer strategy) and the lack of information concerning the internals of predefined components.

Assume-guarantee reasoning [5] is a method of combining the results from the verification of individual components to draw a conclusion about the whole system following certain rules. This has the advantage of avoiding the state explosion problem by not having to actually compose the components, but each component is verified separately.

In this paper we propose a formal verification approach which smoothly integrates with a component based system-level design methodology for embedded systems. Once the model corresponding to the interface logic has been produced, the correctness of the system can be formally verified. The verification is based on the interface properties of the interconnected components and on abstract models of their functionality. Our approach represents a contribution towards increasing both design and verification efficiency in the context of a methodology based on component reuse. This paper mainly demonstrates this methodology on a real-life example (mobile telephone) by verifying three different properties. Each step of the methodology which has to be taken is carefully presented.

2. Methodology Overview

In this paper, we consider systems which are built using predesigned components (IP blocks). Figure 1 illustrates such a system. Each component, in the figures throughout this paper, is depicted as a box with circles on its edge. The circles represent the ports of the component, which it uses for communication with other components.

The *glue logic* inserted between communicating components is depicted in Figure 1 as clouds. For example, the interfaces of two or more components connecting to each

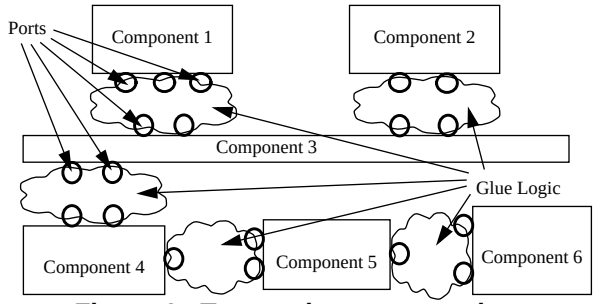


Figure 1. Targeted system topology

other may use different incompatible communication protocols. Thus, they cannot communicate directly with each other. For that reason, it is necessary to insert an adaptation mechanism between the components in order to bridge this gap. This adaptation mechanism would then be the glue logic.

2.1 Objective and Assumptions

The objective of the proposed methodology is to verify the glue logic so that it satisfies the requirements imposed by the connected components.

The methodology is based on the following assumptions:

- The components themselves are already verified.
- The components have some requirements on their environment associated to them, expressed in a formal notation.

The first assumption states that the components themselves are already verified by their providers, so they are considered to be correct. What remains to be verified is the glue logic and the interaction between the components through the glue logic.

According to the second assumption, the components impose certain requirements on their environment. These requirements have to be satisfied in order for the component to function correctly. The requirements are expressed by formulas in a formal temporal logic, in terms of the ports in a specific interface. It is important to note that these formulas do not describe the behaviour of the component itself, but they describe how the component requires the rest of the system (its surrounding) to behave in order to work correctly. In this work, we use (timed) Computation Tree Logic, (T)CTL [5] for expressing these requirements. However, other similar logics may be used as well.

(eq. 1) provides an example of a CTL formula being a constraint associated to a component. This example, also shown in Figure 2, is taken from a design using a connection-based protocol for communication. It consists of a component wanting to send a message to another component at regular time intervals. The sending component is not aware of the connection-based protocol and consequently all its messages must pass through a third component called Protocol Adapter. The protocol adapter implements the chosen protocol and was supplied and verified by a provider.

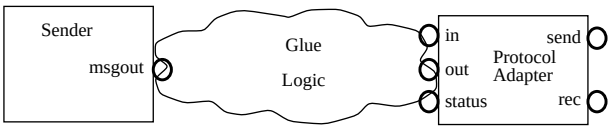


Figure 2. A concrete example of a situation where the methodology can be applied

However, the protocol adapter needs an explicit command to connect to the receiving component, and messages to be sent must be preceded by a particular send command. Such commands are received through port in. The adapter moreover provides the glue logic with information about the connection status through port $status$. Receiving messages arrive through port out. It is the task of the glue logic to supply the adapter with the appropriate commands and to take care of the messages produced by it.

(eq. 1) is associated to the protocol adapter stating that *it is forbidden to send any message unless first connected.*

$$\mathbf{AG}\ ((status = \text{disconnected} \vee \text{init}) \rightarrow \mathbf{A}\ [status = \text{connected}\ \mathbf{R}\ \neg in = \langle \text{send}, \_ \rangle]) \quad \text{(eq. 1)}$$

A set of such formulas is, as mentioned previously, associated to each interface of every component.

2.2 Performing the Verification

The glue logic inserted between two components is to be verified so that it satisfies the requirements imposed by the connected components. Figure 3 illustrates the basic procedure. Model checking is used as the underlying verification technique. The model of the glue logic together with models corresponding to the interface behaviour of the interconnected components (called *stubs*) are given to the model checker together with the (T)CTL formulas describing the properties to be verified.

A stub is a model which behaves exactly in the same way as the component with respect to the interface consisting of ports connected to the glue logic under verification.

As a result of the verification, the model checker replies whether the properties are satisfied in the model or not. If they are not, the model checker provides a diagnostic trace in order to tell the designer what caused the properties to be unsatisfied.

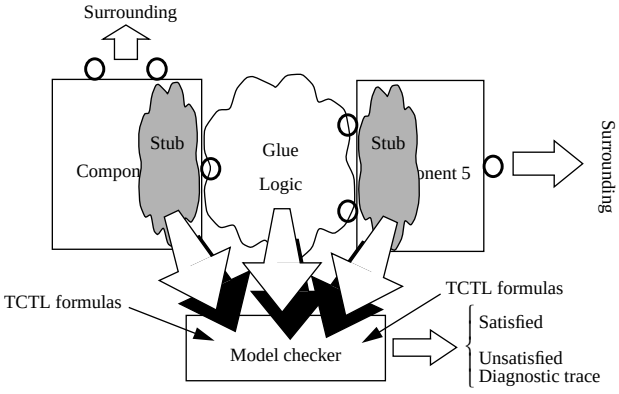


Figure 3. Overview of the proposed methodology

As shown in Figure 3, the part of the system not included in the verification of the particular glue logic is called the *surrounding* of the glue logic.

3. The design representation: PRES+

In the discussion throughout this paper, as well as in the toolset we have implemented, the glue logic, the stubs and the components are assumed to be modelled in a design representation called *Petri-net based Representation for Embedded Systems* (PRES+) [6]. It is a Petri-net based representation with the extensions listed below. Figure 4 shows an example of a PRES+ model.

1. Each token has a value and a timestamp associated to it.
2. Each transition has a function and a time delay interval associated to it. When a transition fires, the value of the new token is computed by the function, using the values of the tokens which enabled the transition as arguments. The timestamp is increased by an arbitrary value from the time delay interval. In Figure 4, the functions are marked on the outgoing edges from the transitions.
3. The PRES+ net is forced to be safe, i.e. one place can at most accommodate one token. A token in an output place of a transition disables the transition.
4. The transitions may have guards. A transition can only be enabled if the value of its guard is true (transitions t_4 and t_5).

It should be pointed out that this design representation is not required by the methodology itself, but the algorithms mentioned here have been developed considering PRES+ as the formal representation. However, if another design representation is found more suitable for a particular design, similar algorithms based on the same ideas can be developed for that design representation.

4. Formal Verification with Stubs

In this section, we will concentrate on two possible scenarios and their slightly different approaches. Either the stubs are given by the component provider or they have to be generated by the designer.

4.1 Stubs are given by the provider

Remember that a stub is a model which behaves exactly in the same way as a certain component with respect to a particular interface. Furthermore, since a component generally has several interfaces, it also has several stubs, one for each interface.

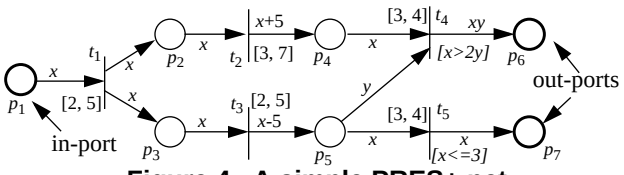


Figure 4. A simple PRES+ net

An interface is defined as a set of ports. Hence, interfaces can be partially ordered with respect to the subset relation. As a consequence, stubs can also be partially ordered by the same relation, due to the fact that they are defined with respect to a particular interface. In the bottom of the hierarchy, so called *empty stubs*[1] can be found. Such stubs either produce or consume (depending on whether the stub will be attached to an out-port or an in-port) events (tokens) containing any message (value) at any point in time, i.e. they behave completely randomly. On the other extreme, in the top of the hierarchy the top-level stub is found. It behaves exactly in the same way as the full component, since it is defined with respect to all ports of the component.

When verifying a particular glue logic using stubs given by the component provider, the designer has to select appropriate stubs from this hierarchy. Theoretical results [7] have shown that if a property, belonging to the sublogic ACTL[2], is satisfied using stubs near the bottom of the hierarchy (low level), it is guaranteed that the property is also satisfied using any higher level stub, including the full component.

Experiments have shown [8] that using low-level stubs generally leads to considerably shorter verification times. However, using low-level stubs makes it more likely that the property is unsatisfied than if high-level stubs were used.

Based on these facts, an iterative approach is suggested. Low-level stubs should initially be used in the verification. If the property is unsatisfied, a stub situated at a higher level in the hierarchy is used instead in the next verification round. The diagnostic trace obtained from the previous verification indicates which stubs violated the property and thus should be changed. This procedure iterates until either the property is verified to true, or the diagnostic trace indicated that the fault is situated in the glue logic (in which case a design error was found). It should be noted that the designer is always guided by the diagnostic trace.

Besides leading to shorter verification times, this methodology also provides a means to perform the verification even if the most appropriate stubs are not available.

4.2 Stubs are generated

In the second scenario when no stubs are given by the component providers, they must be generated by the designer. An algorithm which automatically generates stubs, given the model of a component and an interface, has been developed [8].

The automatically generated stubs do not correspond to the definition in Section 2.2 in the following sense: They are able to produce the same events as the corresponding com-

[1] The name comes from the fact that they somehow correspond to the empty interface, which does not contain any ports at all.

[2] ACTL is a sublogic of CTL, which only allows universal path quantifiers and negation only in front of atomic propositions. All CTL formulas in this paper are ACTL.

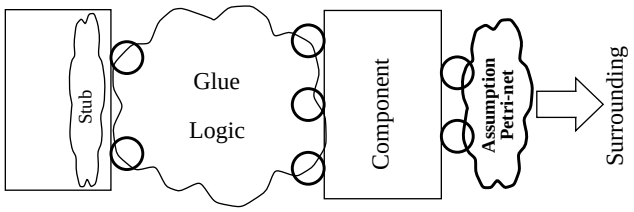


Figure 5. Including the assumptions on the surrounding into the verification process

ponents with respect to the particular interface, but they, in addition, produce more events not produced by the corresponding component. Such stubs are called *pessimistic*.

Due to similar theoretical results as discussed in Section 4.1, if a particular ACTL property is satisfied using pessimistic stubs, it is guaranteed that the property is also satisfied with the full component.

Consequently, it is sufficient to verify the system using pessimistic stubs. However, if the property was not satisfied with the pessimistic stubs, it is necessary to reduce their pessimism, i.e. reduce the number of events produced by the stub which cannot be produced by the full component [8].

During the verification process, the pessimism in the stubs is iteratively reduced until either the property is satisfied or the fault is found to be situated in the glue logic (design fault). It should be noted that this entire process, both pessimism reduction as well as identifying the stub to reduce pessimism on, is guided by the diagnostic trace from the previous verification round.

Sometimes, it might be the case that the information captured by the models of the components or stubs is not sufficient in order to perform the verification. These are cases in which the correct functionality of the involved components depends on certain assumptions relative to those inputs which are connected to the surrounding and, thus, have been ignored in the verification process so far.

An algorithm which generates a model which is able to produce all possible events consistent with a particular ACTL formula, has been developed [8]. A model consistent with the properties on the interfaces connecting to the surrounding, is thus created using this algorithm and attached to the model under verification as indicated in Figure 5. Also in this situation, the diagnostic trace guides the designer to which properties need to be included in the model of the surrounding. The verification process iteratively adds properties to the surrounding until either the property is satisfied or the diagnostic trace indicated a fault in the glue logic.

5. An Illustrative Case Study

The presented verification methodology gives a powerful means to verify large systems using a divide and conquer approach. We have implemented an environment which allows all the activities implied by the methodology to be performed automatically. The only action which needs interaction with the human designer is the examination of the diagnostic trace. The diagnostic trace guides the designer in deciding what action to be taken next.

This section provides an example in order to demonstrate how a system can be verified using the methodology.

5.1 The Mobile Telephone System

The application used as an example is a mobile telephone. Figure 6 shows an overview picture of the model and how the components forming the model are connected. It consists of seven components communicating via an AMBA bus.

1. Microphone. The microphone sends voice data to the transmitter.
2. Buttons. When dialling, the buttons component sends information about which buttons were pressed to the controller.
3. Speaker. The speaker receives voice signals from the receiver and converts it to sound.
4. Display. The display shows on a small screen information sent to it by the controller.
5. Receiver. The receiver receives data from the base-station of the mobile telephone network and passes it on to the designated component.
6. Transmitter. The transmitter receives data from other components in the telephone and passes it on to the base-station.
7. Controller. The controller coordinates the tasks of the other components.

As mentioned previously, these components are supposed to communicate over an AMBA bus. However, since the AMBA bus imposes a certain protocol and the components are not designed for that protocol, glue logics adapting the components to this protocol are inserted. These glue logics are formally verified in subsequent sections.

A few of the components which are directly involved in the example are explained in more detail in the following sections.

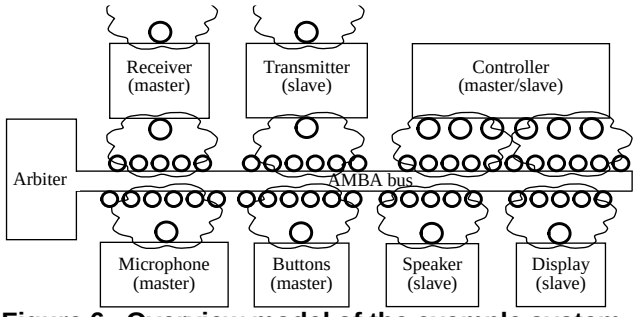


Figure 6. Overview model of the example system, a mobile telephone

5.1.1 Buttons and Display. The peripheral components, such as Buttons and Display, which are used to interact with the end user, are modelled in a simplistic way as shown in Figure 7.

In this example, we assume that the telephone has eleven buttons: the numbers 0 to 9 plus the button ”enter”. When the end user wants to dial a number, he enters the number, presses the button ”enter”, after which the telephone tries to satisfy the request. From the point of view of the component Buttons, the buttons can be pressed in any order at any time. This is modelled by a transition with time delay interval $[0..\infty]$ and the function ”random value from the set $\{0..9, \text{enter}\}$”. The Buttons component has no idea about the semantics of each button being pressed. It is the task of the controller to determine what should happen when a particular button is pressed.

The situation is similar but reverse for Display. Display receives commands about what to show on its screen. In Petri-net terms, this means that tokens in its port are consumed as they appear. The time delay interval depends on how fast the information is processed by the component. In this example, it is assumed that the information is immediately taken care of, i.e. the time delay interval is $[0..0]$.

5.1.2 Controller. The controller component keeps track of what is happening in the system and acts accordingly. Figure 8 shows a model of the component.

Places *accbutton* and *noaccbutton* are marked when the controller is able or is not able to process button data respectively. The data is simply discarded if it is not immediately accepted. Transitions ct_1 to ct_4 take care of this functionality. The transitions have guards so that different actions can be taken depending on which button was pressed. This model only makes a difference between if a number was pressed, $b \in \{0..9\}$, or if ”enter” was pressed, b = enter. When dialling a number, signals (tokens) are also sent in order to update the display. Having pressed ”enter” the telephone number is sent to the transmitter.

Places *calling* and *nocall* record whether a phone call is taking place or not. Transition ct_5 therefore updates these places when a phone call is to be made. Transition ct_7 takes care of incoming phone calls and ct_8 and ct_9 handle the end of a call.

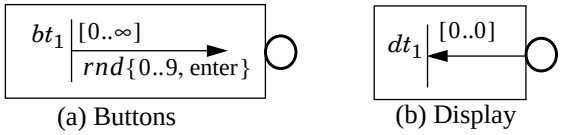
5.1.3 AMBA Bus. All components communicate through an AMBA bus [11]. The AMBA bus consists of two parts, Arbiter and Bus. Figure 9 and Figure 10 introduce the PRES+ models of these two parts. The components communicating over the bus are furthermore divided into two categories, master and slave. Figure 6 indicates to which category each component in the example belongs. Components sending messages are masters and components receiving messages are slaves.

Any master wanting to send data on the bus must first request access to it from the arbiter by emitting the signal HREQBUS. The arbiter will eventually grant access (HGRANT) to any master requesting it, and at the same time, avoid starvation. Once a master is granted access, it may send one bunch of data every clock cycle (time unit, in terms of PRES+). All bunches do not necessarily have to address the same slave. When sending the last bunch, the master notifies this by emitting the signal HTRANS.

However, if a slave is not ready to receive, it is able to put the transaction on hold, or in AMBA bus terms *split*, (HRESP) until it eventually becomes ready (HREADY). During the time period when it is not yet ready to receive, the arbiter might give the access to the bus to another requesting master. When the slave declares itself ready to receive the split data, the master on hold is automatically granted access to the bus again.

The AMBA bus actually consists of two buses, one address bus and one data bus. When a master sends a bunch of data on the bus, it sends the address of the receiving slave on the address bus and the data on the data bus.

Figure 9 shows a part of the model of the arbiter corresponding to one particular master. The part in the figure is



(a) Buttons (b) Display

Figure 7. Models of Buttons and Display

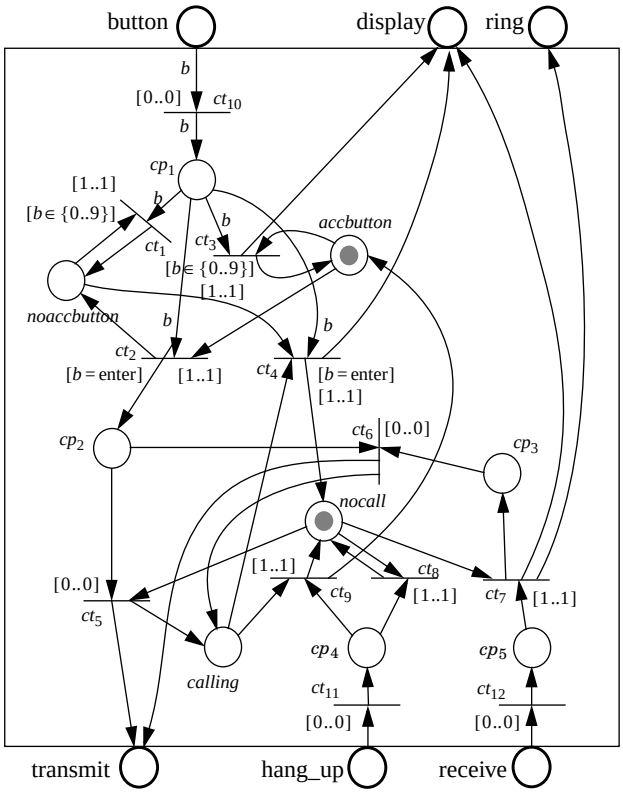


Figure 8. Model of the Controller component

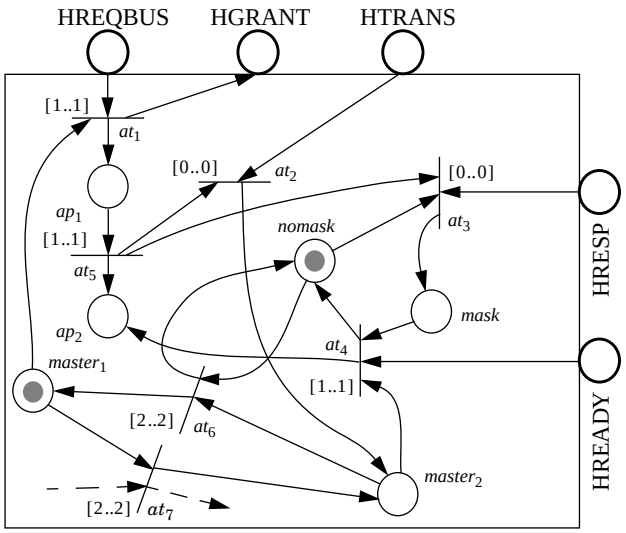


Figure 9. Model of the Arbiter component

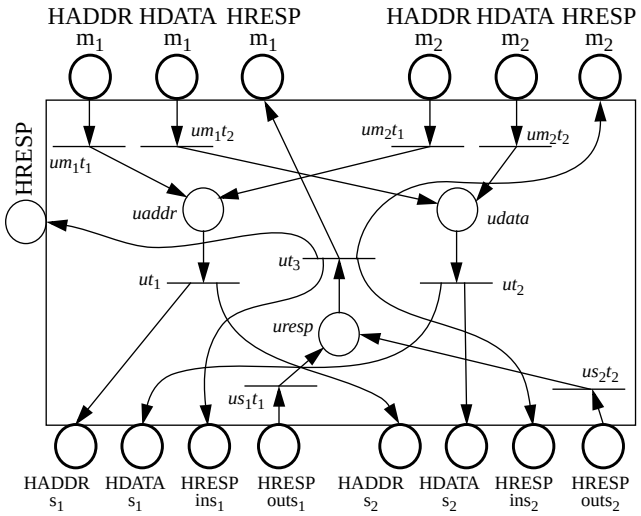


Figure 10. Model of the Bus component

copied once for each master. Places $master_x$ represent which master currently holds the token in a round-robin schedule. The master holding the token has the opportunity to get access to the bus. If a request has not arrived from that particular master, the token moves to the place corresponding to the next master, at_6. Place *mask* is marked when a slave has split the transaction of that master. *nomask* is marked otherwise.

The bus itself just distributes tokens sent to it to all components connected to it. Figure 10 shows a model of the Bus component. All transitions have time delay interval $[0..0]$ and transition function identity. Consequently, it distributes exactly the same token to the rest of the components in zero time.

Port HRESP is directly connected to the arbiter through the port with the same name.

5.1.4 Glue Logics. As has been shown, the components do not contain any functionality to communicate with and over the bus. For this reason, it is necessary to adapt the components and insert a glue logic between the component and the bus.

This design principally contains two types of glue logic, one for handling the master functionality and one for handling the slave functionality for each type of component respectively. Consequently, the glue logics which are situated between the bus and a slave component (see Figure 6) is a slave functionality glue logic, whereas the glue logics situated between the bus and a master component is a master functionality glue logic.

5.2 Verification of the Model

We illustrate the verification of three properties:

1. The controller only receives legal values for button.
   AG $(button \rightarrow button \in \{0..9, \text{enter}\})$
2. When a slave has split a transaction, it will be ready again in the future.
   AG (HRESP $\rightarrow$ **AF** HREADY)
3. When a master has been granted access to the bus, it must eventually close the transaction.
   AG (HGRANT $\rightarrow$ **AF** HTRANS)

These properties were verified using the UPPAAL model checking environment [9]. In order to be able to verify PRES+ models using this tool, the PRES+ models are first translated into the modelling language used by UPPAAL, namely Timed Automata [10]. Such a translation is discussed in [6].

5.2.1 Property 1. The first property to be verified states that the controller must only receive legal values for button. The components included in the verification of this property were the controller, arbiter, bus and the slave functionality glue logic, as illustrated in Figure 11. Table 1 presents the result of the different stages in the verification process.

The property was first verified using empty stubs on all components, except the bus for which a stub was generated. The property was not satisfied using this environment since any data could arrive on the HDATA port of the bus, as indicated by the diagnostic trace. It took about 1 second to obtain this result.

Since the property was not satisfied the diagnostic trace must be examined. According to the diagnostic trace, the

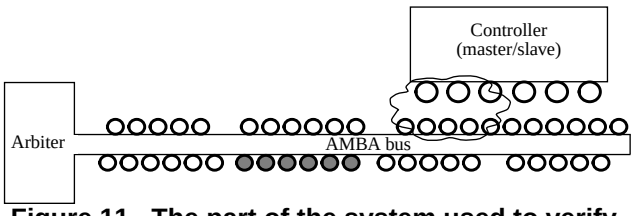


Figure 11. The part of the system used to verify property 1 and property 2

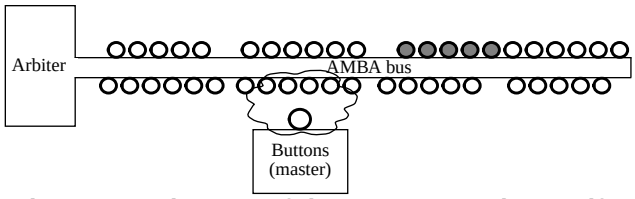


Figure 12. The part of the system used to verify property 3 and the additional assumption of property 1

bus produced a value on port HDATAs_x which is not allowed. In order to do the verification, it was necessary to make an assumption about the surrounding. In this case, it has to be assumed that only data in the set $\{0..9, \text{enter}\}$ can occur in port HDATAm_x. The property is formally given in (eq. 2).

$$\mathbf{AG}\ (\text{HDATA}m_x \rightarrow \text{HDATA}m_x \in \{0..9, \text{enter}\}) \quad \text{(eq. 2)}$$

A Petri-net model for this formula was created together with a new version of the bus stub, now also including port HDATAm_x, and attached to the interface shaded in Figure 11. Using this new stub, the property was satisfied using approximately 2 minutes verification time.

The positive verification result was obtained by making an assumption about the surrounding. In order to finally conclude the positive result, the correctness of the assumption in (eq. 2) must first be established.

The components involved in verifying the assumption itself were the buttons, arbiter, bus and master functionality glue logic, as illustrated in Figure 12. A top-level stub for buttons and empty stubs for the other components was enough for obtaining a result within 7.58 seconds.

5.2.2 Property 2. The second property states that when a slave has split a transaction, it must become ready again in the future. The components included in the verification of this property were controller, arbiter, bus and the slave functionality glue logic, as illustrated in Figure 11. Table 2 presents the result of the different stages in the verification process.

This verification has been started with a faulty glue logic. The fault consisted in that the slave functionality glue logic did not emit HRESP in time (Section 5.1.3). This fault was finally fixed after detection by changing the time delay interval in one transition in the glue logic.

Table 1: Verification results of property 1

Step	Environment	Res	Time
Initial	All empty stubs, except bus generated	false	1.32s
Add assumption on HDATA	All empty stubs, except bus, assumption	true	125.33s
Verify assumption	Buttons top-level stub, other stubs empty	true	7.58s

At first, the property was verified using empty stubs on all components, except that the bus had one generated stub corresponding to interface $\{\text{HRESP}ins_x, \text{HRESP}outs_x, \text{HRESP}\}$. The property was however not satisfied in this environment. The diagnostic trace indicated that messages were sent too quickly on port HADDR and HDATA. In other words, an infinite amount of data was sent in the same clock cycle. In the real system, only one bunch of data can be sent in the same clock cycle. The problem was solved by increasing the level of the stubs on ports HADDR and HDATA from empty to level one stubs. These stubs were given (created manually).

The property was again verified in the updated environment, but it was still not satisfied. The diagnostic trace led to the design error in the glue logic as described previously. After fixing the error, the property was reverified using the very same environment, but still with a negative verification result.

The problem this time was a too pessimistic stub for the bus component. This caused the fact that no signal would ever be emitted on port HREADY. Due to the pessimism in the generated stub, it was exchanged with a given one[1]. After additional 4 minutes, the property was finally satisfied.

5.2.3 Property 3. The third property states that when a master has been granted access to the bus, it must eventually close the transaction. The components included in the verification of this property were the buttons, arbiter, bus and master functionality glue logic, as illustrated in Figure 12. Table 3 presents the result of the different stages in the verification process.

This verification was also started with a faulty glue logic. The fault consisted in that the glue logic could not differentiate whether a particular split request was a result of its own attempts to send or not. The fault was fixed, after detection during verification, by adding a structure to keep track of the necessary information.

Table 2: Verification results of property 2

Step	Environment	Res	Time
Initial	All empty stubs, except {HRESP*in*, HRESP*out*, HRESP}	false	2.47s
Use higher level stubs	Initial except Level 1 stubs for HADDR, HDATA	false	28.39s
Correct design error	Initial except Level 1 stubs for HADDR, HDATA	false	87.57s
Use higher level stubs	Empty stubs for controller and top-level stub for the bus	true	246.14s

[1] Another way to continue the verification would have been to continue with less pessimistic stubs generated automatically and, if needed, with added models corresponding to assumptions on the surrounding.

As with the verification of the previous properties, the first environment used consisted of empty stubs. In this environment, Arbiter may grant access to the bus without it even being requested. Consequently, after such an unrequested grant, data will not be sent and in particular the transaction will not be closed. Thus, the property is not satisfied.

To avoid this problem revealed by the diagnostic trace, the empty stubs of the arbiter were replaced with a given stub. After half a second's verification time, the property proved again unsatisfied. The diagnostic trace shows that the reason was that a transaction can be split, but the slave will never signal after a while that it is ready to receive data again. It is however a requirement on the slaves to eventually signal that they are again ready after a split. Therefore, a Petri-net corresponding to the formula $\mathbf{AG}\,(\mathrm{HRESP} \rightarrow \mathbf{AF}_{\leq 5}\mathrm{HREADY})$ was generated and attached to the bus on the shaded interface in Figure 12. Note that it is not necessary to verify this assumption as it is a requirement of the arbiter and the bus in order to work properly. Besides, the property was already verified in the previous section. Even with this extra assumption the property proved unsatisfied.

The diagnostic trace indicated an error in the glue logic. It did not record whether the split requests were a result of its own attempts to send or not. A mechanism for this was added and the property was reverified with the same environment. After 41 minutes a positive result was obtained.

5.3 Discussion

This section has tried to give an example on how to use the verification methodology presented in this paper, in practice.

The successive steps through the methodology are guided by the diagnostic trace which all the time gives feedback to the user what to do next. It might indicate that too pessimistic stubs were used, that there is an error in the glue logic, or that assumptions regarding the surrounding have to be introduced.

6. Conclusions

This paper has presented a verification methodology which takes advantage of the fact that designs are built using reusable components. The methodology assumes that these components are already verified, and concentrates on the glue logics interconnecting the components. Every component has a number of properties associated to it which it requires the system to satisfy in order to work correctly.

An essential part of this methodology involves models of the component behaviour with respect to a certain interface of the component. These models are called *stubs*.

Two scenarios have been presented: either the stubs are given by the component provider, or they are generated by the designer given the model of the component. Both scenarios can be efficiently exploited in the verification process by adopting an iterative approach. Furthermore, properties regarding the surrounding of the glue logic under verification can also be incorporated into the process.

An example has also been presented in order to demonstrate the feasibility of using the approach on reaalistic designs. It was carefully demonstrated how three different properties were verified on a mobile telephone and how the diagnostic trace guided the designer in each step.

Most of the activities in this verification methodology can be automatically performed and have been implemented in a tool. The only activity which needs interaction with a human designer, is the examination of the diagnostic trace. The diagnostic trace constantly guides the designer in deciding what action to take next.

Table 3: Verification results of property 3

Step	Environment	Res	Time
Initial	All empty stubs	false	0.14s
Use higher level stubs	Initial except arbiter stub	false	0.52s
Add property 2 as assumption	Arbiter stub given, button stub empty, bus with assumption	false	2.58s
Correct design error	Arbiter stub given, button stub empty, bus with assumption	true	2467.42s

7. References

[1] J. Turley, "Embedded Processors by the Numbers", *Embedded Systems Programming*, vol. 12, May 1999.

[2] J. Haase, "Design Methodology for IP Providers", *Proc. DATE*, 1999, pp. 728-732.

[3] D. Gajski, A C.-H. Wu, V. Chaiyakul et al, "Essential Issues for IP Reuse", *Proc. ASP-DAC*, 2000, pp. 37-42.

[4] R. Seepold, N.M. Madrid, A. Vörg et al, "A Qualification Platform for Design Reuse", *Proc. ISQED*, 2002, pp. 75-80.

[5] E.M. Clarke, O. Grumberg, D.A. Peled, "Model Checking", *The MIT Press*, 1999.

[6] L.A. Cortés, P. Eles, Z. Peng, "Verification of Embedded Systems using a Petri Net based Representation", in *Proc. ISSS*, 2000, pp. 149-155.

[7] D. Karlsson, P. Eles, Z. Peng, "Formal Verification in a Component Reuse Methodology", in *Proc. ISSS*, 2002, pp. 156-161

[8] D. Karlsson, "Towards Formal Verification in a Component-based Reuse Methodology", Licentiate Thesis No 1058, Linköping Studies in Science and Technology, http://www.ep.liu.se/lic/science_technology/10/58/

[9] UPPAAL homepage: http://www.uppaal.com

[10] E.M. Clarke Jr, O. Grumberg, D.A. Peled, "Model Checking", MIT Press, 1999.

[11] A. Roychoudhury, T. Mitra, S.R. Karri, "Using formal techniques to Debug the AMBA System-on-Chip Bus Protocol", in *Proc. DATE*, 2003, pp. 828-833.